SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

Confined Types in Java

Jan Vitek[1,*] and Boris Bokowski[2,†]

[1] *CERIAS, Computer Sciences Department, Purdue University,*
[2] *Object Technology International, Ottawa, CA*

SUMMARY

The sharing and transfer of references in object-oriented languages is difficult to control. Without any constraint, practical experience has shown that even carefully engineered object-oriented code can be brittle, and subtle security deficiencies can go unnoticed. In this paper, we present inexpensive syntactic constraints that strengthen encapsulation by imposing static restrictions on the spread of references. In particular, we introduce *confined types* to impose a static scoping discipline on dynamic references and *anonymous methods* to loosen confinement somewhat to allow code reuse. We have implemented a verifier which performs a modular analysis of Java programs and provides a static guarantee that confinement is respected. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: sharing, aliasing, Java, object-orientation, security

1. INTRODUCTION

Writing secure code is hard. The steady stream of security defects reported in production code attests to the difficulty of the task. Software systems, such as the Java virtual machine, that permit untrusted code to mingle with authorized code raise the stakes for security as trust boundaries become thinner and fuzzier.

In this paper, we focus on the interaction of sharing and security in object-oriented programming languages and propose a solution tailored for Java. In Java, like most modern object-oriented languages, objects are manipulated exclusively through references. Basic operations such as assignment and parameter passing will thus create aliases. Consequently, controlling the spread and sharing of object references is difficult. Pervasive aliasing implies that there can be no accurate notion of ownership – veryfing *a priori* if an object is reachable from another is undecidable (static analysis is limited to conservative results [1]). In a partially

*Correspondence to: J. Vitek, CERIAS, Computer Sciences Department, Purdue University, West Lafayette, IN 47907, USA
E-mail: jv@cs.purdue.edu
†Part of this work was done while at the Freie Universität Berlin, Berlin, Germany.

Received 13 April 1999
Revised 3 August 2000

trusted environment, this means that any method may be called from untrusted code forcing developers to program defensively. For example, to be safe, each method that accesses or updates sensitive information must somehow verify that it is invoked by a trusted context. In Java these checks are performed at run-time. Such dynamic checks not only impose a run-time penalty but may also cause programs to fail during execution. So, for efficiency and reliability it is preferable to shift the burden of verification to an earlier stage of the program life-cycle. In this paper we focus on compile-time checks, other solutions are discussed in related work.

Practitioners are faced with a tension between security and efficiency. At one end of the spectrum automatically inserted checks before every sensitive instruction – assuming such instructions can be identified – could render a system secure but would lead to dismal program performance. In practice, Java programs are secured by interspersing checks in the program logic. The choice of which operations to guard left to the developer [2, 3]. Unfortunately with such *ad hoc* approaches, nothing short of full-fledged program verification will ensure that no check has been omitted running the risk of compromising the security of the entire system.

Reusability, one of the major benefits of the object-orientation, creates its own set of problems for security. Just as with concurrency which gives rise to the well-known inheritance anomaly [4], inheriting code from classes with different security policies may create security anomalies. A class that is safe in one context may open a security hole when extended in another context. Of course, from the point of view of the library designer, it is not feasible to design classes that are secure in all contexts and, even if one could the performance of the library would almost surely be unacceptable.

At least some of the difficulty involved in engineering secure code can be traced to aliasing and to the lack of clear interfaces between trust domains. If references to a trusted object can be acquired by untrusted code then this object must be secured against attacks. Unfortunately, as we stand now a Java virtual machine manages objects of different protection domains — code loaded from different sources — but imposes no boundaries between these domains. So, from the security engineer's viewpoint, there is no distinction between objects that may be obtained by an adversary and secure ones. Thus there is no well-identified place to put security checks.

One solution to this quandary is to separate objects that are internal to a protection domain (for some definition of domain) from objects that may be accessible directly from the outside. Internal objects can be code to implement a given behavior without concern to security, while the remaining objects are the interface to other domains and must implement a security policy. Such a separation of concerns is efficient since the core of the system can be written without security checks. Moreover, it has the potential to improve security since a smaller set of classes, the interface objects, become the focal point for security analysis. Current object-oriented languages do not provide the means to enforce such a distinction between objects. Static access modifiers restrict how certain object types are manipulated [5] — curtailing visibility of methods and fields — and restrict the scope of types, but there is no encapsulation of references [6]. As we will show in later examples the interplay of inheritance, subtyping and reference semantics makes it quite hard to control the scope of references and thus to prevent them from being leaked to untrusted code.

We propose *confined types* for Java as an aid for writing secure code. One way of thinking about confined types is as a machine checkable programming discipline that prevents leaks of sensitive object references. We are not proposing a change to the language, rather we are suggesting guidelines how to use Java's existing facilities to enforce the desired encapsulation property. Given some definition of a protection domain, we say that a type is *confined* to that domain if all references to objects of that type originate from within the domain. In other words, code outside of the domain should never be allowed to manipulate confined objects directly. Confinement differs from existing access control mechanisms in that it constrains access to object references rather than classes. The difference is most visible when considering subtyping. Class-based restrictions (such as the Java `private` keyword) can be circumvented by casting the protected object to one of its unrestricted supertypes. With confined types this is not allowed. For all practical purposes, confined types should be viewed as enforcing static scoping on dynamic object references.

We have implemented a confinement checker for the Java programming language with `CoffeeStrainer` [7], a framework for static checking of structural constraints on Java programs. This implementation uses Java packages as protection domains. Packages are well suited for the task since they group logically related classes and provide the basic access control mechanisms that we need. The impact on the language is minimal. We extend Java with two modifiers, one for classes (`confined`) and one for methods (`anon`). In our implementation these annotation are embedded in comments for backwards compatibility with existing Java compilers. While certain programming tasks may be clumsier with confined types, we argue that these restrictions are mild and that reasoning about security is much simpler. One significant aspect of the proposal is that the constraints are checked statically. Thus there is no extra runt-ime overhead and confinement is guaranteed to hold at run-time — avoiding the need to worry about "confinement breach" exceptions. Furthermore our confinement checker is modular, working one class at a time, and only requires access to the confining package. Outside code does not have to be checked and dynamic loading is supported. Because the annotations do not affect program semantics, a valid program with confinement annotations exhibits behavior identical to the same program without annotations.

Road Map. The paper is organized as follows. We start by reviewing language-based protection mechanisms in Section 2. In Section 3, we argue that these mechanisms are not sufficient using the example of a well-known Java security defect. Section 4 introduces one part of the solution, anonymous methods, which, while independent from confined types, are essential to allow confined code that inherits from library classes. We present confined types in Section 5 and give a complete programming example in Section 6. Related work is discussed in Section 7. We conclude with design choices, implications for genericity, and applications to software engineering.

2. PROGRAMMING LANGUAGE SECURITY

Security is turning into a software issue as the mechanisms used to implement security policies are cheaper and more flexible in software than in hardware [8, 9, 10]. Security is often discussed in terms of principals, objects, protection domains and security policies. We briefly introduce these terms (see Gollman [11] for more complete definitions). Principals are the entities whose actions must be controlled. Principals invoke operations on objects. Here, the term object is used in more general sense than in object-oriented programming. An object may be a datum, a file, a hardware device, *etc.* The context within which a principal executes is called a protection domain. Access to resources within the same protection domain is not checked, while cross-domain operations must be authorized by a security policy. Implementing security policies at the programming language level has been advocated for three main reasons. Firstly, language semantics can help to reason about program behavior and thus to prove security properties. Secondly, type systems and static analysis algorithms can reduce the run-time cost of security. Finally, protection domains can be made lightweight and allow fine-grained interactions.

2.1. Safe Languages

Safe programming languages guarantee that the execution of programs proceeds without overrunning memory, that types are not misinterpreted and data is not mistaken for executable code. In Java, safety depends on four techniques: *bytecode verification* to ensure that programs are well-formed, *strong typing* to guarantee that values are used according to their definition, *automatic memory management* to prevent errors such as deleting a live object, and *memory protection* to prevent array and stack operations from overflowing [12]. While safety is not the same thing as security it is an essential foundation for the latter [13].

2.2. Information flow control

Over the last 20 years an abundant body of work has been devoted to information flow control. Multilevel security policies [14], originally conceived for military applications, are based on the notion that all data is labeled with security levels and that principals may only access data for which they have security clearance. The objective being to guarantee *non-interference* — a property which, informally, means that the values of low level security variables may not depend on high level security variables [15, 16, 17]. This requires checking all channels of communication that may create information flows (these include implicit channels such as conditional expressions and loops, as well as the more exotic timing and probabilistic channels). To date these techniques are still not used in practice. Part of the problem stems from inherent restrictions; to achieve non-interference in a multi-threaded language Volpano and Smith [18] had to forbid the guards of loops from depending on high security variables. Forbidding loops in sensitive code is quite stringent, yet not sufficient since some probabilistic channels remain. A more fundamental problem with information flow control is that it assumes a homogeneous software system in which security labels are set once and for all, and all subsystems agree on the labels and on their meaning. In a distributed system assembled from heterogeneous components these assumptions do not hold as there are as many policies as protection domains (*e.g.*, applets), each of which may decide what data is sensitive. Some of these problems have been addressed in a sequential subset of Java [19], but extending the approach to the full language is still an open problem. To summarize, information flow provides a sound basis for building secure systems, but current technology remains too restrictive for widespread usage.

2.3. Access Control Policies

Discretionary access control mechanisms do not provide the same strong guarantees as information flow control but are easier to use in practice. The idea is to perform security checks before any potentially dangerous operation in order to verify that the current program has the authority to perform the requested action. Schemes such as *capabilities* and *access control lists* have been used to implement discretionary access control policies.

Static access control. Object-oriented languages provide two basic means for controlling access to objects. The first is *access modifiers* such as `private`, `protected`, and `public`, to restrict the visibility of attributes and classes. The second is *type abstraction*; abstract types and subsumption can be used to limit the operations that can be invoked on an object [20, 21]. This second approach is not applicable in languages such as Java in which the run-time type of objects can be retrieved by the program (*e.g.*, through the `instanceof` operator or reflection).

Dynamic access control. Java provides dynamic access control mechanisms based on call stack inspection to verify the privileges of the (transitive) caller of the current method [2]. Another scheme is to use objects as capabilities [22] by interposing a restricted proxy object between the user and the target ([23], see also [24, 25, 26]).

2.4. Certified Code

Recently, a number of researchers have investigated the concept of certified code. Proof-Carrying Code (PCC) proposed by Necula and Lee [27, 28] is the most general certification framework. A great variety of properties can be specified in PCC and components need only be bundled with a proof expressed in formal logic to provide assurance. Approaches that do not rely on explicit proofs but rather on strong typing provide a more lightweight alternative to PCC. In type-theoretic solutions the security properties that can be specified over components are determined by the information provided by the type system. Essential properties include language safety [13] as enforced by the byte code verifier in Java [12] or by Typed Assembly Language [29, 30].

2.5. Summary

While information flow policies are too restrictive, neither discretionary access control nor certified code directly provides a solution. Access control mechanisms dependent on dynamic checks are error-prone since it is easy to forget one check. No guarantee can be given that all potentially dangerous operations are protected by access checks. The static access control mechanisms described above were originally conceived for software engineering purposes rather than for security. Not surprisingly they only provide a partial solution. Finally, certified code is undoubtedly a promising research direction, but it requires a substantial commitment from both software providers and users, as well as a substantial implementation effort to reimplement components so that they can be proved secure.

3. A SECURITY BREACH: THE JAVA CLASS SIGNING BUG

In Java, each class object – that is each instance of class `Class` – has a list of signers. These signers are principals under whose authority the class acts. This list is used by the security architecture to determine access rights of the class at run-time. A leak of a reference to this internal data structure was the cause of a security flaw that allowed untrusted applets to gain all access rights in JDK 1.1.

The breach was caused by the conjunction of two seemingly innocuous operations. The first operation is a method of `java.security.IdentityScope` which allows any applet to find out all the principals known to the system, amongst which some are likely to be trusted. The second operation allows a class to get the list of principals that signed it. The method that returned the array of signers erroneously returned an alias. Arrays being mutable data structures, the applet then only needs to update the array with the signature of trusted principals to gain that principal's access rights.

3.1. The Security Breach in Detail

In Figure 1 `signers` is the system's internal array of references to `Identity` object. Updating this array is definitely a sensitive operation but the generic array class has no provisions for checking the authority of the code that manipulates it. The security breach is caused by the `getSigners()` method. The attacker need only invoke `getSigners()` to create an alias to the array and thus be allowed to freely update the signatures. One simple fix for this bug is to return a copy of the array as shown in Figure 2. This solution is *ad hoc* because we have no guarantee that an alias to this object is not leaked by other parts of the package.

It is interesting to note that that none of the standard Java protection mechanisms seem to help. Access modifiers and type abstraction are not relevant here. Restricting the use of the `Identity` objects would do no good since the attack does not interact with `Identity` objects, it only needs to acquire references to them and copy those references. Information flow control does not apply either, since we do want to allow applets to read the signature information and to see identities known to the system. Finally, inserting dynamic checks in the array update operation, which is the point where the security policy is actually broken, is unrealistic since all array updates performed in the JVM would incur the cost of a dynamic check.

```
public class Class {
    private Identity[] signers;
    public Identity[] getSigners( ) {
        return signers;
    }
}
```

Figure 1. Signatures without confined types

```
public class Class {
     private Identity[] signers;
     public Identity[] getSigners( ) {
          Identity[] pub = new Identity[signers.length];
          for (int i = 0; i < signers.length; i++) pub[i] = signers[i];
          return pub;
     }
}
```

Figure 2. Ad-hoc fix of security problem

We now give a solution that guarantees that none of the key data structures used in code signing escape the scope of their defining package. It is interesting to note that code of the solution using confined type is very similar to the *ad hoc* solution of Figure 2. The advantage of course is that confinement is checked automatically, while the correctness of the *ad hoc* solution depends on the programmer.

3.2. Class Signing with Confined Types

To prevent software defects such as the one outlined above, we propose ensuring that *references* to identity objects be confined to the `java.security` package. This can be achieved, for example, by renaming the `Identity` class to `SecureIdentity` and declaring it *confined*. Intuitively, the meaning of confinement is that references to instances of a confined class, or to instances of any of its subclasses, cannot be disclosed to or accessed by other packages. That is to say, only the classes defined in package `java.security` may interact with `SecureIdentity` objects. Figure 3 outlines our solution.

In order to preserve the functionality of the original interface, we define a new class `Identity` which is accessible outside of the security package. This new class acts as a facade: it implements the public methods of `SecureIdentity` and keeps a private reference to a `SecureIdentity` instance. `Identity` encapsulates the real identity object [23, 24]. The `Identity` class is purely for external use, it is neither a subclass nor a superclass of `SecureIdentity` and thus cannot be confused with a `SecureIdentity` object within the security package. Any attempt to return a `SecureIdentity` object to an outside package will be caught at compile-time as a violation of confinement. This solution preserves the functionality of the original program, in fact outside code is not aware of the existence of confined types. But from a security engineering point of view, attention is directed to the `Identity` class since it can be accessed by untrusted components, and may thus (if deemed necessary) include dynamic security checks.

As mentioned above, the code of `getSigners()` is similar to the *ad hoc* fix of Figure 2. What matters here is that it is the confinement rules that forced us to introduce the copy. Thus, it is guaranteed that no leak can go unnoticed. The key to the solution is that the type of `signers` is not related to the return type of the method. Furthermore, since confinement constraints

```
confined class SecureIdentity ... {
     ...  // the original Identity implementation
}

public class Identity {
     SecureIdentity target;
     Identity(SecureIdentity t) { target = t; }
      ...   // public operations on identities;
}

public class Class {
     private SecureIdentity[] signers;
     public Identity[] getSigners( ) {
          Identity[] pub = new Identity[signers.length];
          for (int i = 0; i < signers.length; i++)
               pub[i] = new Identity(signers[i]);
          return pub;
     }
}
```

Figure 3. Signatures with confined types.

extend to arrays, this means that `SecureIdentity[]` is confined as well. `getSigners()` is forced to allocate an unconfined array to which newly created objects of type `Identity` are copied.

Confined types help in developing secure code by drawing a strong demarcation line between internal representation objects and external interface objects. Before introducing confined types, we will present anonymous methods which play a central role for the practicality of our proposal.

4. ANONYMOUS METHODS

An anonymous method is a method that does not reveal the current instance's identity except to other anonymous methods. Therefore, it cannot introduce new aliases to the current instance. Although anonymous methods are essential to allow confined types to inherit methods from unconfined parents, they also have interesting properties in their own right which may be useful in other contexts [31]. For example, in a language like Eiffel anonymous methods can be safely invoked on expanded objects.

In Java parlance, an anonymous method is a non-native method that may use `this` *only* for accessing fields of the current instance, calling other anonymous methods on itself, or for

```
class Example {
     int count;
     int anon ok( A arg ) {
1         alsoOk( arg.foo() );
2         return count ;
     }
     void anon alsoOk( int i ) {
3         count = i + count ;
     }
     Example notOk( A arg ) {
4         arg.bar( this ) ;
5         arg.o = this ;
6         notOk( arg );
7         return this ;
     }
}
```

Figure 4. Anonymous methods.

reference comparisons against other object references. Thus an anonymous method keeps its implicit `this` parameter secret by not assigning `this` to a variable, nor providing `this` as a method argument, nor returning `this` as the method's return value, nor throwing `this` if the method belongs to a subclass of `Throwable`. As a rule of thumb the body of an anonymous method can always be rewritten so that the keyword `this` does not occur, because Java allows implicit occurrences of `this`. There are two exceptions to this rule: The first exception are reference comparisons involving `this`, and the second exception are accesses to fields hidden by a parameter or a local variable of the same name or accesses to fields of a superclass which are shadowed by a field with the same name in a descendant class.

Figure 4 presents a valid class `Example` with two anonymous methods (`ok, alsoOk`) and a non-anonymous method (`notOk`). Lines (`1 - 3`) show examples of anonymity-preserving code, while (`4 - 7`) show examples that do not preserve anonymity. Line (`4`) reveals `this` to method `bar`. (`5`) stores `this` in a field of `arg`. Line (`6`) calls a non-anonymous method (don't mind the infinite recursion). Finally, (`7`) returns `this`.

Because the definition of anonymous methods is recursive, we require anonymous methods to be declared as such explicitly. In our examples, we use the annotation **anon**, while in the implementation we include an `@anon` tag in the method's Javadoc comment for backwards compatibility. Our checker verifies that each method declared anonymous conforms to the definition. In addition to the constraint regarding the use of `this`, there is another constraint regarding anonymity of overridden methods. Since anonymity is a property that potential callers rely on, methods in subclasses that override an anonymous method must be anonymous as well.

We treat constructors as a special case of methods. They can be declared anonymous as well with the same constraints. In Java, the first statement of each constructor is a call to another constructor, which may be in the same class, or in the direct superclass of the current class. Without an explicit call, the constructor of the superclass is called implicitly. An anonymous constructor must thus ensure that explicit and implicit calls are made only to anonymous constructors. The `Object` constructor, the only one that does not call another constructor, is anonymous by definition, as are several other commonly used methods in `Object`: `wait()`, `notify()`, `notifyAll()`, `hashCode()`, and `finalize()`.

We summarize the constraints that apply to anonymous methods and constructors:

$\mathcal{A}1$ – **The reference `this` can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparisons.**

$\mathcal{A}2$ – **Anonymity of methods and constructors must be preserved in subtypes.**

$\mathcal{A}3$ – **Constructors called from an anonymous constructor must be anonymous.**

$\mathcal{A}4$ – **Native methods may not be declared anonymous.**

Clearly, anonymous methods rule out some perfectly safe programs. It is important to assess how restrictive our proposal actually is and whether common programming idioms would become too cumbersome to be practical or too inefficient. For instance, the *visitor pattern*n breaks anonymity to implement a form of double dispatching [32].

To obtain a better sense of the impact of anonymity declarations on programming style, we analyzed JDK 1.1 to find out how many existing methods meet the above mentioned criteria ($\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$). The data has been collected by iterating a static analysis detecting anonymity violations. In each iteration, methods flagged by the analysis were declared as `non-anon`. The process was repeated until a fixpoint was reached. The results, summarized in Table I, are encouraging. Without changes to existing code, between 84% and 95% of the methods are already anonymous. With some care a portion of those non-anon methods could be re-written to become anonymous.

Package	`java.util`	`java.awt`
classes + interfaces	28 + 3	63 + 7
all methods	351	1246
anon methods	332 (95%)	1047 (84%)

Table I. Anonymous methods in existing code.

Anonymous methods are related to Boyland's concept of borrowed receiver [33] and Leino's captures [34]. Boyland defines a reference to be borrowed by a method if the method can not store the reference and thus does not introduce any static aliases. Anonymous methods are simpler more specialized instantiations of these concepts. Section 5.4 explains our use of anonymous methods.

5. CONFINED TYPES

Objects of a confined type may not be referenced or accessed from outside of the type's protection domain. Confined types are introduced by annotating class or interface definitions with the keyword `confined`. (In the implementation, confined types are declared by implementing an empty interface (`implement Confined`) for backwards compatibility.) Instances of confined types are called *confined objects*. We adopt packages as protection domains to take advantage of access modifiers. Instances of confined classes may thus only be referenced or accessed from within a single package. Confinement also applies to subclasses of a confined class. We can unambiguously refer to an object's *confining package*, meaning the package in which the object's class is defined. We can also refer to the package of a confined type since all classes (or interfaces) that extend (implement) a confined class (interface) belong to the same package. Figure 5 summarizes the relationships between an object `obj` in package `outside` and the objects `conf` and `unconf` from package `inside`. A reference from `obj` to the confined object is not allowed, but all other references, including from `conf` to objects outside of the package are allowed.

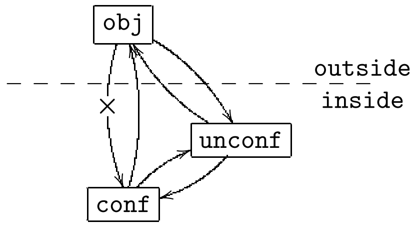

Figure 5. References between packages.

It is important to understand that we are not trying to prevent information from leaking through covert channels, just to prevent references to confined objects from being transferred out of their confining package.

5.1. Overview of the Problem

Before listing the confinement constraints, it is helpful to consider all constructs with which object references may be transferred from a package `inside` to another package `outside`. These points are illustrated in Figure 6. Each line labeled `r1` to `r10` demonstrates a reference transfer.

```
package inside;
public class C extends outside.B {
     void putReferences() {
          C c = new C();
r1        outside.B.c1 = c;
r2        outside.B.storeReference(c);
r3        outside.B.c3s = new C[] {c};
r4        calledByConfined();
r5        implementedInSubclass();
r6        throw new E();
     }
     void implementedInSubclass() { }
r7   public static C f = new C();
r8   public static void C m() {
          return new C();  }
r9   public static C[] fs = new C[]{new C()};
r10  public C() { }
}
public class E extends RuntimeException { }

package outside;
public class B {
r1   static inside.C c1;
r2   static void storeReference(inside.C c2) { // store c2 }
r3   static inside.C[] c3s;
r4   void calledByConfined() { // store this }
     static void getReferences() {
r7        inside.C c7 = inside.C.f;
r8        inside.C c8 = inside.C.m();
r9        inside.C[] c9s = inside.C.fs;
r10       inside.C c10 = new inside.C();
          D d = new D();
          try { d.putReferences();
r6        } catch (inside.E ex) { // store ex }
     }
}
class D extends inside.C {
r5   void implementedInSubclass() { // store this }
}
```

Figure 6. Transferring references.

We start with reference transfers that originate from code of package `inside`. The possible targets in package `outside` fall into three categories: fields, method and constructor parameters (including the implicit parameter `this`), and parameters of catch clauses. Taking into account that object references can be stored in arrays, we distinguish six cases for transfers from `inside`:

`r1` Package `inside` assigns a reference to one of its objects to a field in package `outside`,
`r2` Package `inside` calls a method or constructor defined in package `outside` passing a reference to one of its objects as an argument,
`r3` Package `inside` wraps an object reference into an array (or multiple nested arrays) and uses points `r1` or `r2` for transferring the array reference,
`r4` Calling a method or constructor defined in a class in package `outside` from a subclass of that class in package `inside` (the implicit parameter `this` is transferred),
`r5` Calling a method defined in a class in package `outside` from a superclass of that class in package `inside` (the implicit parameter `this` is transferred),
`r6` Package `inside` throws an exception which is handled by a catch clause defined in package `outside` (the exception object is transferred).

We now list reference transfers that originate in package `outside`. The possible sources in package `inside` fall into three categories: fields, method return values, and references to newly instantiated objects using the operator `new`. Again, taking into account that object references can be stored in arrays, we distinguish four cases for reference transfers originating in package `outside`:

`r7` Package `outside` reads a field of package `inside` containing a reference to an instance of a class defined in package `inside`,
`r8` Package `outside` calls a method of package `inside` that returns an object reference to an instance of a class defined in package `inside`,
`r9` Package `outside` uses points `r7` or `r8` to obtain a reference to an array (or multiple nested arrays), into which package `inside` has wrapped an object reference,
`r10` Package `outside` instantiates an object of a class defined in package `inside` using the `new` operator.

We now introduce the constraints that prevent reference transfers. The presentation proceeds as follows: Section 5.2 gives constraints on class and interface declarations. Section 5.3 presents constraints that prevent widening. Section 5.4 discusses constraints that deal with hidden widening. Based on the constraints introduced so far, Section 5.5 explains why reference transfers originating in the inside package cannot occur. Finally, Section 5.6 presents the remaining constraints, which address reference transfers originating in outside packages.

5.2. Confinement in Declarations

The first two constraints restrict the declaration of classes and interfaces. The goal is to ensure that confined types are only visible in their package and to guarantee that subtyping preserves confinement.

$\mathcal{C}1$ – **A confined class or interface must not be declared public and must not belong to the unnamed global package.**

$\mathcal{C}2$ – **Subtypes of a confined type must be confined as well.**

$\mathcal{C}1$ ensures that confined types have package-local access. Confined types cannot belong to the unnamed global package, since this package is "open" to extensions. $\mathcal{C}2$ guarantees that if a confined class (or interface) is extended (implemented) then the extending class (interface) is also confined. Thus, the confinement property extends transitively to all subtypes of a confined type. Note that $\mathcal{C}1$ and $\mathcal{C}2$ together imply that confined types may only have subtypes in the same package.

5.3. Preventing Widening

To prevent references to confined objects from escaping their package, reference widening from a confined type to an unconfined supertype cannot be allowed. Clearly, the root of the type hierarchy, `java.lang.Object`, is not confined. Thus, if a confined reference can be widened and stored in an `Object` variable, then the confined object may leak out of its package[†]. In Java, reference widening may occur in either of:

- an assignment, if the declared type of the left hand side of the assignment is a supertype of the assigned expression's static type,
- a method call, if the declared type of a parameter is a supertype of the corresponding argument expression's static type,
- a return statement, if the declared result type of the method is a supertype of the result expression's static type,
- a cast expression, if the target type of cast is a supertype of the expression's static type.

Widening must be prevented if it entails losing the confinement property of an object reference. The following constraint enforces confinement.

$\mathcal{C}3$ – **Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.**

As noted in Section 3, Java arrays are a way to leak references as well. Consequently, the constraint takes arrays into account as well. For a confined type `A`, we regard the array type `A[]` to be a confined type as well, called a *confined array type*, so that they are a special case of $\mathcal{C}3$. In general, the constraints imply that confined objects may not be stored in unconfined collections (of which arrays are just one example). Although this restricts common programming styles,

[†]Note that widening a reference so that its type is `Object` cannot be allowed because this already enables the attack of our motivating example in Section 3. An object reference whose static type is `Object` can be stored in an array whose static type is `Object[]` if the dynamic types of the object and the array match.

the signed classes example showed that it is exactly this kind of potential leakage which is easy to overlook. Thus, we think it is worth the effort to provide special-purpose confined collections (or arrays) rather than trading security for the reuse of collection classes. Section 8.2 discusses the impact of confined types on genericity.

5.4. Preventing Hidden Widening

In addition to the obvious widening of the previous section, implicit or *hidden* widening occurs whenever a method inherited from an unconfined superclass is invoked on a confined object. Upon entry in the inherited method the implicit parameter `this` which refers to the current instance is widened from the confined type to the unconfined supertype.

Clearly, hidden widening should not be ruled out completely, since this would make it impossible to derive confined classes from non-trivial unconfined classes. But allowing confined classes to extend unconfined classes without restrictions is dangerous. The reference to the current instance may leak out if a method in the superclass transfers it to any other object. However, anonymous methods of Section 4 are safe since they do not leak `this`. We can now give the constraints that ensure the safety of hidden widening. We say that methods *defined* by a class are the (newly introduced or overridden) methods appearing in its body; all other methods are *inherited.*

$\mathcal{C}4$ – **Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.**

$\mathcal{C}5$ – **Constructors called from the constructors of a confined class must either be defined by a confined class or be anonymous constructors.**

$\mathcal{C}4$ constrains inherited methods, in the case of overridden methods, *i.e.*, if a method defined in a superclass is overridden in a confined subclass, it is safe to execute the method since it preserves confinement. Similar to methods, constructors of unconfined superclasses that are called by the constructors of a confined class need to be anonymous. This applies to instance field initializers and instance initialization blocks as well, since these might also leak out a reference to the object.

We should emphasize that these constraints need only be checked within the defining package of the confined type since it is not possible to invoke methods of confined types of another package. Also, note that methods and constructors defined by confined classes need not be anonymous. Moreover, note that interfaces do not play a role here since they do not introduce code. Anonymous methods ease the restrictions that would otherwise be imposed on inheritance. Without them, it would be unsafe to invoke any inherited method of a confined object.

5.5. Preventing Transfer from the Inside

In our list, points `r1` to `r6` involve transfers that originate in the inside package. Based on the constraints introduced so far, points `r1` and `r2` — assigning to a field in an outside package, and passing parameters to a method in an outside package — are not allowed for confined types. Since neither a confined type itself nor one of its subtypes is accessible from the outside package (due to constraints $\mathcal{C}1$ and $\mathcal{C}2$), the type of the field or parameter can only be an unconfined supertype of the confined type. But then, transferring the reference would require reference widening which is ruled out by constraint $\mathcal{C}3$.

Similarly, point `r3` — wrapping references to confined objects in an array and transferring the array reference by assigning it to a field or passing it as a parameter — is not possible, because arrays of confined types are confined as well.

Reference transfers according to point `r4` — calling a method in an unconfined supertype — are not ruled out completely; rather, constraints $\mathcal{C}4$ and $\mathcal{C}5$ require the called methods (resp. constructors) to be anonymous, as discussed in Section 4. Thus, it is possible to transfer references, but only to code that can neither discloses the reference to a non-anonymous method nor depends on the reference.

Item `r5` — transferring `this` to a subclass by calling a method which is implemented in the subclass — cannot transfer a confined reference to an outside package, because constraints $\mathcal{C}1$ and $\mathcal{C}2$ make sure that all subclasses of a confined type must reside in the same package as the confined type.

With Java exceptions, there is another opportunity for transferring references which is rather obscure: If an exception of a certain type is thrown, it may be caught with a catch clause whose formal parameter is of a supertype of the actual exception that was thrown. Since we don't see important uses where exception objects should be confined to a package, we just disallow subtypes of `java.lang.Throwable` to be confined types, thus disallowing reference transfers according to point `r6`. The class `java.lang.Thread` also requires special treatment since one of its static methods returns a reference to the currently executing thread object. We require that:

$\mathcal{C}6$ – **Subtypes of `java.lang.Throwable` and `java.lang.Thread` may not be confined.**

We now consider reference transfers that may be initiated outside of the confining package.

5.6. Preventing Transfer from the Outside

Reference transfers from the outside package to the inside (`r7` – `r10`) have not yet been addressed. They involve transfers that originate in some outside package. The new constraints are:

$\mathcal{C}7$ – **The declared type of public and protected fields in unconfined types may not be confined.**

$\mathcal{C}8$ – **The return type of public and protected methods in unconfined types may not be confined.**

Fields whose declared types are confined types should not be accessible from outside the package, *i.e.*, confined fields in accessible (unconfined) types may not be public or protected ($\mathcal{C}7$), preventing object reference transfer according to point `r7`. Although the confined type itself is not accessible from outside the package, confinement is not enforced in other packages.

Thus, if a field of a confined type was accessible, it would be possible for the outside package to widen the reference to an unconfined supertype.

By similar reasoning, methods in unconfined types which return a confined type should not be accessible from outside the package, *i.e.*, no method returning a confined type should be public or protected ($\mathcal{C}8$). Thus, point `r8` is prevented as well. Again, note that confined array types are a special case of the general constraint, so fields of confined array types and methods returning confined arrays must have private or package-local access, preventing `r9`. Instantiating a confined class from outside (point `r10`) cannot occur because confined classes are not accessible from outside.

5.7. Implementation

We use CoffeeStrainer [7] to statically check confinement and anonymity constraints. Modularity, support for dynamic loading and backwards compatibility are important design goal for our confinement checker. Modularity is addressed by design of the constraint rules — we have chosen rules that can be checked locally, one class at a time. Support for dynamic loading is addressed by categorizing packages in three groups: **(1)** packages containing confined types (*confining packages*), **(2)** packages containing unconfined superclasses of confined types (*extended packages*), **(3)** and plain packages (*outside packages*). For confining packages confined classes must be checked for constraints $\mathcal{C}1$ and $\mathcal{C}3$ – $\mathcal{C}6$, all other classes in a confining package must abide by rules $\mathcal{C}2$ – $\mathcal{C}4$ and $\mathcal{C}6$ – $\mathcal{C}8$ (and possibly $\mathcal{A}1$ – $\mathcal{A}4$ if they have anonymous methods.). In extended packages, we need only check that methods declared anonymous follow the rules $\mathcal{A}1$ – $\mathcal{A}4$. Finally, plain packages need not be checked at all. These checks can be performed at compile time or postponed until load time. Figure 7 illustrates these different checks. Dynamic loading is allowed in extended and plain packages (1,2). For confining packages, dynamic loading must be restricted to prevent untrusted code from being added to a trusted package. This restriction can implementing by using sealed JAR files [35, 36] or by instrumenting directly the class loader. Our current implementation does not address generic classes and nested classes.We are investigating extensions to the rules to cover both.

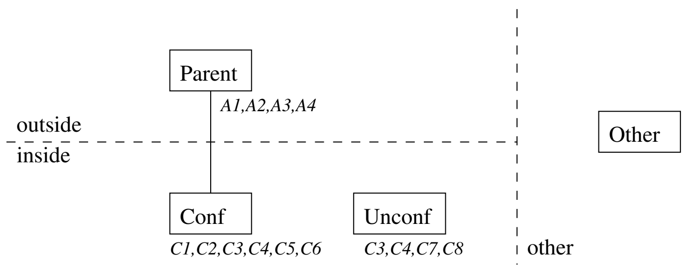

Figure 7. The confined class `Conf` in package `inside` extends `Parent` in package `outside`. `inside` is a confining package, `outside` is extended by a confining package, and `other` is a plain package. The constraints to be checked are indicated below each class.

6. USING CONFINED TYPES: PUBLIC-KEY CRYPTOGRAPHY

Public-key cryptography is one of the essential tools for security in distributed systems. The implementation of public key cryptography must therefore be secure. Furthermore it should be reusable. More specifically, our goal is to ensure that the random number objects that generate keys should not be accessible outside of the implementation of the RSA algorithm [37]. Further, we would like to offer the guarantee to clients of the RSA package that the object that represent their private keys remain confined to their application and that under no circumstance untrusted code be granted access to a private key.

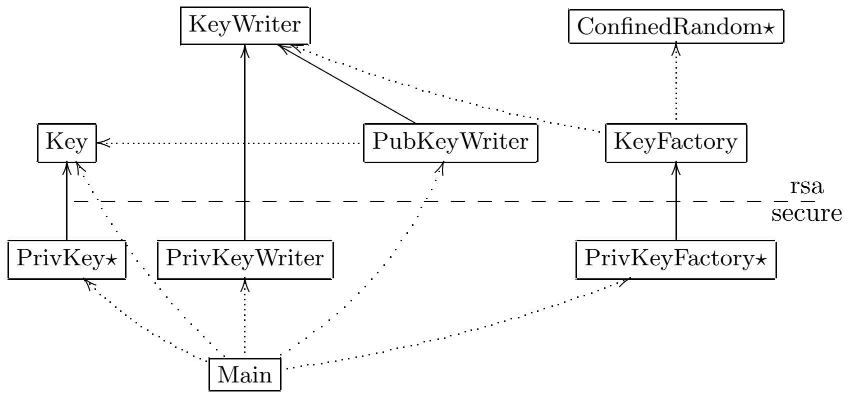

Figure 8. Relationships between package `rsa` and package `secure`. Full arrows indicate subtyping relations. Dashed arrows indicate implementation dependencies. Confinement is denoted by a ⋆.

We use confined types to achieve the desired security properties. It is noteworthy that the solution requires little effort on the part of the client (the users) of the RSA library. We structure the code in two packages:

- Package `rsa`: a reusable public-key cryptographic library.
- Package `secure`: one particular user of the `rsa` package.

The classes that we want to protect are `ConfinedRandom`, the random number generator, and `PrivKey`, the actual private keys. The first class belongs to the `rsa` implementation and the second is owned by the client of the library, the `secure` package. `ConfinedRandom` is confined in package `rsa`, while `PrivKey` is confined in `secure`. Public keys are implemented by the `Key` class and are not confined since clients may want to pass them around to other packages. Of course nothing prevents a client package from making public keys confined as well.

The package `rsa`, Figure 9, provides a class `Key` that encapsulates RSA encryption. Class `KeyFactory` generates a key pair (`pub`, `priv`) such that a message encrypted with the public key can be decrypted using the private key and vice versa, *i.e.*, `pub.crypt(priv.crypt (m))` returns `m`. The implementation of `KeyFactory` relies on class `ConfinedRandom` for generating the keys.

```
package rsa;

import java.math.BigDecimal;
import java.util.Random;

public class Key {
     public BigDecimal mod;
     public BigDecimal exp;

     anon public String crypt(String msg) { /* return (msg^^exp)%mod */ }
}

confined class ConfinedRandom extends Random { }

public interface KeyWriter {
     anon public void setValues( BigDecimal m, BigDecimal e);
}

public class KeyFactory {

     private ConfinedRandom randomGenerator = new ConfinedRandom(
                         System.currentTimeMillis());

     anon public void genKeyPair( KeyWriter pub, KeyWriter priv) {
          // set internal values of both key objects,
          // using random generator...
     }
}

public class PubKeyWriter implements KeyWriter {
     private Key key;

     public PubKeyWriter(Key k) { key = k; }

     anon public void setValues( BigDecimal m, BigDecimal e) {
          key.mod = m;
          key.exp = e;
     }
}
```

Figure 9. Package containing RSA algorithm

The package `secure`, Figure 10, introduces classes `PrivKeyFactory` and `PrivKey` to, respectively, generate and represent private keys. A class `Main` is given to demonstrate how keys are used. There are several other classes in the implementation, we will detail them in the following paragraphs. Figure 8 illustrates the relationships between the two packages. Relevant portions of the implementations of both packages are given in Figure 9 and Figure 10.

In class `Key` the fields `mod` and `exp` are `public`. Although this allows access to sensitive information from the outside, a reference to a key is required to read the fields' values. The idea is to subclass `Key` in another package and to make this subclass confined. Accordingly, the method `crypt` is declared `anon` since otherwise this method could not be called on a confined object ($\mathcal{C}4$).

Often confined types require only a trivial implementation, as can be seen in class `ConfinedRandom`. This is an example of making an unconfined class confined in another package by subclassing. The class `ConfinedRandom` is used in class `KeyFactory` for the field `randomGenerator`. This field is declared `private` so that only the class `KeyFactory` has to be reviewed by the programmer for potential leakage of a reference to the random generator object or leakage of its internal state.

The class `KeyFactory` does not set the internal values of Key objects directly. Rather, it uses the interface `KeyWriter` which normally would not appear in a design without confined types. The reason for this is that both `Key` and `KeyFactory` will be subclassed and made confined in another package. If `KeyFactory` referenced `Key` directly, the confined subclass of `Key` could not be used with `KeyFactory` or a subclass of it because at some place a reference widening to the original type `Key` would be needed, which is forbidden by $\mathcal{C}3$. Class `PubKeyWriter` trivially implements the interface `KeyWriter`.

Note also that `PrivKey` does not define any new methods or fields. However, a new implementation of `KeyWriter` is needed for accessing the internal values of the confined type `PrivKey`. Due to constraint $\mathcal{C}3$, which prevents widening from `PrivKey` to `Key`, the previously defined class `PubKeyWriter` cannot be used. The similarity of the new implementation `PrivKeyWriter` to `PubKeyWriter` suggests that genericity would help here; this is discussed in Section 8.2.

Similar to `PrivKey`, a confined subclass `PrivKeyFactory` is derived from `KeyFactory`. The interesting point here is that the superclass has access, and uses, a confined class (namely `ConfinedRandom`), but our restrictions guarantee that these values can not be leaked to the subclass.

In class `Main`, a private and a public key object is created. Note that private or package-local access for field `privateKey` is required by $\mathcal{C}7$, while `publicKey` can be `public`. In `main()`, then, a `Factory` object is created and `genKeyPair()` is invoked on it, providing two instances, one of `PubKeyWriter` and one of `PrivKeyWriter`.

```
package secure;

import rsa.*;
import java.math.BigDecimal;

confined class PrivKey extends Key { }

private class PrivKeyWriter implements KeyWriter {
     private PrivKey key;

     public PrivKeyWriter(PrivKey k) { key = k; }
     anon public void setValues(
               BigDecimal m, BigDecimal e) {
          key.mod = m;
          key.exp = e;
     }
}

confined class PrivKeyFactory extends KeyFactory { }

public class Main {
     private static PrivKey privateKey = new PrivKey();

     public static Key publicKey = new Key();

     public static void main(String[] args) {
          PrivKeyFactory keyFactory = new PrivKeyFactory();
          keyFactory.genKeyPair(
               new PubKeyWriter(publicKey),
               new PrivKeyWriter(privateKey));
          // use keys for encryption and decryption...
     }
}
```

Figure 10. Confining a type in a different package

7. Related Work

The original impetus for the work presented here comes from difficulties of implementing secure and reliable systems in Java. Some of these difficulties can be attributed to aliasing [38, 26]. Confined types follow up on work on flexible alias protection [6] in which we tried to control aliasing at the level of individual objects. Related work can be divided into literature on alias control and on security; we review both topics in the following two subsections.

7.1. Alias Control

Reference semantics permeate object-oriented programming languages, it is thus not surprising that the issue of controlling aliasing has been the focus of numerous papers in the recent years [39, 40, 34, 41, 6, 42, 43, 44]. We will discuss briefly the most relevant works.

In [6], we proposed flexible alias protection to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing mode declarations specify constraints on sharing of references. The mode `rep` protects *representation objects* from exposure. In essence, `rep` objects belong to a single owner object and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode `arg` marks argument objects which do not belong to the current owner, and therefore may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles. In [44], Clarke, Potter, and Noble formalize representation containment by means of ownership types. Both papers have been presented in the context of a simple programming language without inheritance or subtyping. There is no obvious way to maintain containment in the presence of either. Confined types were designed to support both concepts.

Hogg's Islands [40] and Almeida's Balloons [41] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from [6] is that both proposals strive for full encapsulation, that is, all objects reachable from an owner are protected from aliasing. This is equivalent to declaring everything inside an Island or Balloon as `rep`. This is restrictive, since it prevents many common programming styles: it is not possible to mix protected and unprotected objects as done with flexible alias protection and confined types. Hogg's proposal extends Smalltalk-80 with sharing annotations but it has neither been implemented nor been formally validated. Almeida did implement an abstract interpretation algorithm for deciding whether a class meets his balloon invariants. But his approach requires whole-program analysis. The constraints presented in this paper can be checked modularly, one class at a time.

The Sandwich types of Genius, Trapp, and Zimmermann [42] are a compromise between flexible alias protection and balloons. The objects protected from aliasing are computed by inspection of the type graph of the whole program. The criterion for protection is when a type is only reachable from another (owner) type. The prototypical example is the class `LIST_CELL` which only appears in the implementation of `LIST`. The drawback of sandwich types is that they require global program analysis, and do not deal with inheritance and subtyping.

	Lang.	Inherit.	Encaps.	Enforce	Modularity	Granularity
Islands [40]	Sm-80	Yes	Full	Static	Class	Object
Balloons [41]	Toy	Yes	Full	Static	Whole-prg.	Object
Flexible Alias [6]	Toy	No	Partial	Static	Class	Object
Sandwich [42]	Toy	No	Full	Static	Whole-prg.	Class
Kent [43]	Eiffel	Yes	Partial	Dynamic	–	Object
Confined types	**Java**	**Yes**	**Partial**	**Static**	**Class**	**Package**

Table II. Comparison of alias control techniques.

Finally, Kent and Maung [43] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time. Confined type are static, a choice better suited to security since errors are caught earlier.

Table II compares the proposals discussed above. Partial encapsulation allows selective protection of components. Enforcement of constraints can either be done at compile-time (*static*) or at run-time (*dynamic*). Verification can require analysis of the entire program (*whole-program*) or be modular at the class level (*class*). Granularity of protection can be either: at the *object* level, meaning that individual objects are protected, at the *class* level, meaning that all instance of a class are treated as a single encapsulation domain, and finally at the *package* level, meaning that all instances of all classes belonging to the same package are grouped in a single domain.

7.2. Security

Confined types depart from the work on information flow control [45, 19, 17]. We are not trying to protect the information content of objects, as shown by the class signing example of Section 3, rather we control the flow of language level objects, or more precisely, object references. Further, confined types are as much about integrity as secrecy.

The elegant paper of Leroy and Rouaix [20] has similar goals as the work presented in this paper. The authors formalize the security properties of applets written in a strongly typed programming language. They propose a technique based on type abstraction to guarantee that certain locations in the store will not be written by untrusted components. Leroy and Rouaix did not deal with subtyping or inheritance. They chose a simple functional language (an idealization of Caml), our work can be viewed as an extension of theirs to object-oriented languages.

Another recurrent theme is the use of objects as *capabilities* or guards [23, 46, 24]. Different variants of this scheme boil down to the facade pattern [32] in which a facade object protects access to one or more targets. The facade implements the security policy for access to the targets. The proposals typically do not provide any strong security guarantees, since some reference to one of the targets may still be leaked to an adversary. Confined types strengthen this approach. If target objects are confined, then no reference can be revealed to outside code.

8. Discussion

8.1. Design alternatives

Unlike flexible aliasing protection [6], our proposal protects entire packages. This flat protection model can be limiting. First, the objects we want to protect need not all be in the same package. Second, it is not possible to compose larger systems out of components.

We have considered different designs allowing a class to be confined to a group of classes which need not be in the same package. For example, we could define the notion of a reference protection domain, then each class would be declared to belong to some domain. The following declaration bundles three classes in a protection domain.

```
domain java.security.Identity, java.lang.Class,
     java.security.SecureIdentity;
```

The `SecureIdentity` class is still defined as confined, but now it will be visible only to the other two classes in the domain. The drawback of external domains is that we cannot use package visibility to define methods that may only be used by classes in the same domain.

This idea could be extended further to hierarchical protection domains. This requires named domains. Next we define two domains, one is the aforementioned domain, the second is a larger domain encompassing all security classes.

```
domain IdentityDomain is
     java.security.Identity, java.lang.Class,
     java.security.SecureIdentity;

domain SecurityDomain is
     java.security.*, IdentityDomain;
```

While possible, hierarchical domains are pushing towards more complex models such as flexible alias protection [6, 44]. The cost in complexity may outweigh the gains.

8.2. Confined Types and Genericity

As has already been noted in sections 5.3 and 6, confined types could profit from parameterized types. Because parameterized types reduce the need for reference widening (*e.g.*, when storing objects in collections), much more reuse would be possible if confined types were combined with parameterized types. Interestingly, we found that confined types may influence the ongoing discussion about how to incorporate genericity in Java because they do not fit equally well with all proposals that have been put forward so far. There are two observations:

The first observation concerns the translation scheme used to translate generic types to normal classes and interfaces so that they can be executed on unmodified Java virtual machines. With a homogenous translation scheme [47, 48], different instantiations of a parameterized type are translated to a single class or interface. Because parameterized types instantiated with a confined type then cannot be distinguished at run-time from those instantiated with unconfined types, references to confined objects could leak out by confusing them with references to unconfined objects. Thus, confined types fit better with proposals that have a heterogenous translation scheme [49, 50], in which different instantiations of parameterized types are translated to different classes or interfaces.

When looking at the example presented in Section 6, another observation for the discussion about genericity can be made: In the example, the two classes `Key` and `KeyFactory` had to be decoupled by the intermediate interface `KeyWriter`. Although this interface would not be needed in a conventional design, the decoupling was required for subclassing both `Key` and `KeyFactory` in package `secure`. This suggests that virtual types [51] might be a better fit for confined types, since they allow subclassing of a whole family of classes in such a way that use relationships between classes in the original family become use relationships between classes in the derived family.

8.3. Software Engineering

Confined types may be useful from a software engineering point of view as well. Confined types can be viewed as the representational components of a framework which cannot be accessed from the outside. The external interface of the framework would then consist of unconfined types that usually do not contain functional code but make up a facade [32] through which the framework must be used. Based on this architecture, a package designer may decide to change the interface of a confined type, knowing that the effects of that change are limited to the single package and will not break client code.

Note that unlike techniques such as guards and capabilities (see Section 7.2), in which every possible access path to otherwise unprotected objects needs to be controlled, confined types take the opposite approach. The default is to disallow any direct access to confined types, and then facades may be used to grant access for certain uses.

8.4. Optimization

Confined types can help program optimization. Since the scope of a confined type is limited to a package, aggressive optimizations can be applied within the package. For instance, static analysis of the package code contains all uses of that package's confined types. It may thus be possible to remove methods that are not called in the package, since they are dead code, and even modify the structure of confined objects or of the class hierarchy [52].

Restricting widening improves the precision of concrete type inference and thus helps generating better code for confined types.

Finally, Genius, Trapp, and Zimmermann have shown that aliasing restrictions can be used to improve locality of memory access and have obtained significant speed up on small scale programs [42].

9. Conclusion

Software security is a difficult problem. This paper introduces two new language mechanisms, confined types and anonymous methods, that can be used for controlling the dissemination of object references. This control eases the task of writing secure code, since the interface between components is sharper.

Confinement and anonymity are enforced by a set of syntactic constraints which can be verified statically. Thus, our proposal incurs no run-time overhead and all confinement violations are caught before running the program.

We have implemented a confinement verifier for Java using CoffeeStrainer [7]. The verification procedure is modular since classes are analyzed individually. Our extensions are transparent, annotated classes can be compiled by the standard Java compiler.

Acknowledgments

The authors whish to thank John Boyland, Doug Lea, James Noble, Jens Palsberg, Philip Wadler, the participants of the Intercontinental Workshop on Aliasing in Object Oriented Systems, and the anonymous reviewers for their comments and suggestions. An earlier version of this paper was presented at the OOPSLA'99 conference.

REFERENCES

1. Landi W. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1992; **1**(4).
2. Gong L. *Inside the Java 2 Platform Security Architecture: Cryptography, APIs, and Implementations.* The Java Series. Addison-Wesley Pub Co, 1999.
3. Tardo J, Valente L. Mobile agent security and Telescript. *IEEE CompCon*, 1996.
4. Matsuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha PW, Yonezawa A (eds.). The MIT Press, 1993, chapter 4; 107–150.
5. Gosling J, Joy B, Steele GL. *The Java Language Specification.* The Java Series. Addison-Wesley: Reading, MA, USA, 1996.
6. Noble J, Potter J, Vitek J. Flexible alias protection. *Proceedings of ECOOP'98*, vol. 1543 of *LNCS*. Springer-Verlag: Brussels, Belgium, 1998.
7. Bokowski B. CoffeeStrainer: Statically-checked constraints on the definition and use of types in Java. *Proceedings of ESEC/FSE'99*, 1999.
8. Chase J, Levy H, Baker-Harvey M, Lazowska E. Opal: A single address space system for 64-bit architectures. *Proceedings of the Fourth Workshop on Workstation Operating Systems*, 1993; 80–85.
9. Grimm R, Bershad BN. Security for extensible systems. *Proceedings of 6th Workshop on Hot Topics in Operating Sytems*, 1997; 62–66.
10. Lucco S, Sharp O, Wahbe R. Omniware: A Universal Substrate for Web Programming. *World Wide Web Journal*, 1995; **1**(1):359–368.
11. Gollman D. *Computer Security.* John Wiley & Sons, 1999.
12. Yellin F. Low level security in Java. *Fourth International Conference on the World-Wide Web*, 1995.
13. Kozen D. Language-based security. Technical Report TR99-1751, Cornell University, Computer Science, 1999.
14. Denning D. A lattice model of secure information flow. *Communications of the ACM*, 1976; **19(5)**:236–243.
15. Volpano D, Smith G. Confinement properties for programming languages. *SIGACT News*, 1998; **29**(3):33–42.

16. McLean J. Security models. In *Encyclopedia of Software Engineering*, Marciniak J (ed.). Wiley & Sons, 1994.
17. Volpano D, Smith G. A type-based approach to program security. *Proceedings TAPSOFT'97*, vol. 1214 of *LNCS*. Springer-Verlag: Berlin, Germany, 1997.
18. Smith G, Volpano D. Secure information flow in a multi-threaded imperative language. *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998; 355–364.
19. Myers AC. Jflow: Practical static information flow control. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, 1999.
20. Leroy X, Rouaix F. Security properties of typed applets. *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998; 391–403.
21. Riecke JG, Stone CA. Privacy via Subsumption. *Fifth Workshop on Foundations of Object-Oriented Languages*, 1998.
22. Levy H (ed.). *Capability Based Computer Systems*. Digital Press, 1984.
23. Gong L. Guarding objects. *Mobile Agents and Security*, Vigna G (ed.), vol. 576 of *LNCS*. Springer-Verlag: Berlin, Germany, 1998; 1–23.
24. Hagimont D, Mossière J, de Pina XR, Saunier F. Hidden software capabilities. *16th International Conference on Distributed Computing System*. IEEE CS Press: Hong Kong, 1996.
25. Wallach D, Balfanz D, Dean D, Felten E. Extensible Security Architectures for Java. *Proceedings of the 16th Symposium on Operating System Principles*, 1997.
26. Vitek J, Bryce C. The JavaSeal mobile agent kernel. *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, 1999.
27. Necula GC. Proof-carrying code. *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997; 106–119.
28. Necula GC, Lee P. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, Vigna G (ed.), vol. 1419 of *LNCS*. SV, 1998; 61–91.
29. Morrisett G, Crary K, Glew N, Walker D. Stack-based typed assembly language. *Second International Workshop on Types in Compilation*, Leroy X, Ohori A (eds.), vol. 1473 of *LNCS*. Springer-Verlag: Kyoto, 1998; 95–117.
30. Morrisett G, Walker D, Crary K, Glew N. From System F to Typed Assembly Language. *Twenty-fifth ACM Symposium on Principles of Programming Languages*, 1998; 85–97.
31. Boyland J. Deferring destruction when reading unique variables. Technical report, University of Wisconsin – Milwaukee, 1999.
32. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
33. Boyland J. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 2000. In this issue.
34. Detlefs D, Rustan K, Leino M, Nelson G. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.
35. Microsystems S. Support for extensions and applications in the version 1.2 of the Java platform. 2000.
36. Zaks A, Feldman V, Aizikowitz N. Sealed calls in Java packages. *OOPSLA '2000 Conference Proceedings*, ACM SIGPLAN Notices. ACM, 2000.
37. Rivest R, Shamir A, Aldeman L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 1978; **21**(2).
38. Vitek J, Serrano M, Thanos D. Security and communication in mobile object systems. In *Objects at Large*, Tsichritzis D (ed.). University of Geneva, 1997.
39. Hogg J, Lea D, Wills A, de Champeaux D, Holt R. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 1992; **3**(2).
40. Hogg J. Islands: Aliasing Protection in Object-Oriented Languages. *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991; 271–285. Published as ACM SIGPLAN Notices, volume 26, number 11.
41. Almeida PS. Balloon types: Controlling sharing of state in data types. *ECOOP'97—Object-Oriented Programming, 11th European Conference*, Aksit M, Matsuoka S (eds.), vol. 1241 of *LNCS*. Springer-Verlag: Jyväskylä, Finland, 1997; 32–59.
42. Genius D, Trapp M, Zimmermann W. An approach to improve locality using Sandwich Types. *Proceedings of the 2nd Types in Compilation workshop*, vol. LNCS 1473. Springer Verlag: Kyoto, Japan, 1998.
43. Kent S, Maung I. Encapsulation and Aggregation. *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*. Prentice Hall, 1995.

44. Clarke DG, Potter JM, Noble J. Ownership types for flexible alias protection. *OOPSLA '98 Conference Proceedings*, vol. 33(10) of *ACM SIGPLAN Notices*. ACM, 1998; 48–64.
45. Heintze N, Riecke JG. The SLam calculus: Programming with secrecy and integrity. *Proceedings of the 25th POPL*, 1998.
46. Hawblitzel C, Chang CC, Czajkowski G, Hu D, von Eicken T. Implementing Multiple Protection Domains in Java. Technical Report 97-1660, Cornell University, Department of Computer Science, 1997.
47. Odersky M, Wadler P. Pizza into Java: Translating theory into practice. *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.
48. Bracha G, Odersky M, Stoutamire D, Wadler P. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA Proceedings*. ACM Press: Vancouver, BC, 1998.
49. Myers A, Bank J, Liskov B. Parameterized types for Java. In *POPL Proceedings*. ACM Press: Paris, France, 1997.
50. Bokowski B, Dahm M. Poor man's genericity for Java. In *JIT Proceedings*. Springer-Verlag: Frankfurt, Germany, 1998.
51. Thorup KK, Torgersen M. Unifying genericity – combining the benefits of virtual types and parameterized classes. In *ECOOP'99*, LNCS. Springer-Verlag: Lisbon, Portugal, 1999.
52. Tip F, Laffra C, Sweeney PF, Streeter D. Size matters: Reducing the size of Java class file archives. Technical report, IBM Research Report RC 21321, 1998.